

汇编语言程序设计

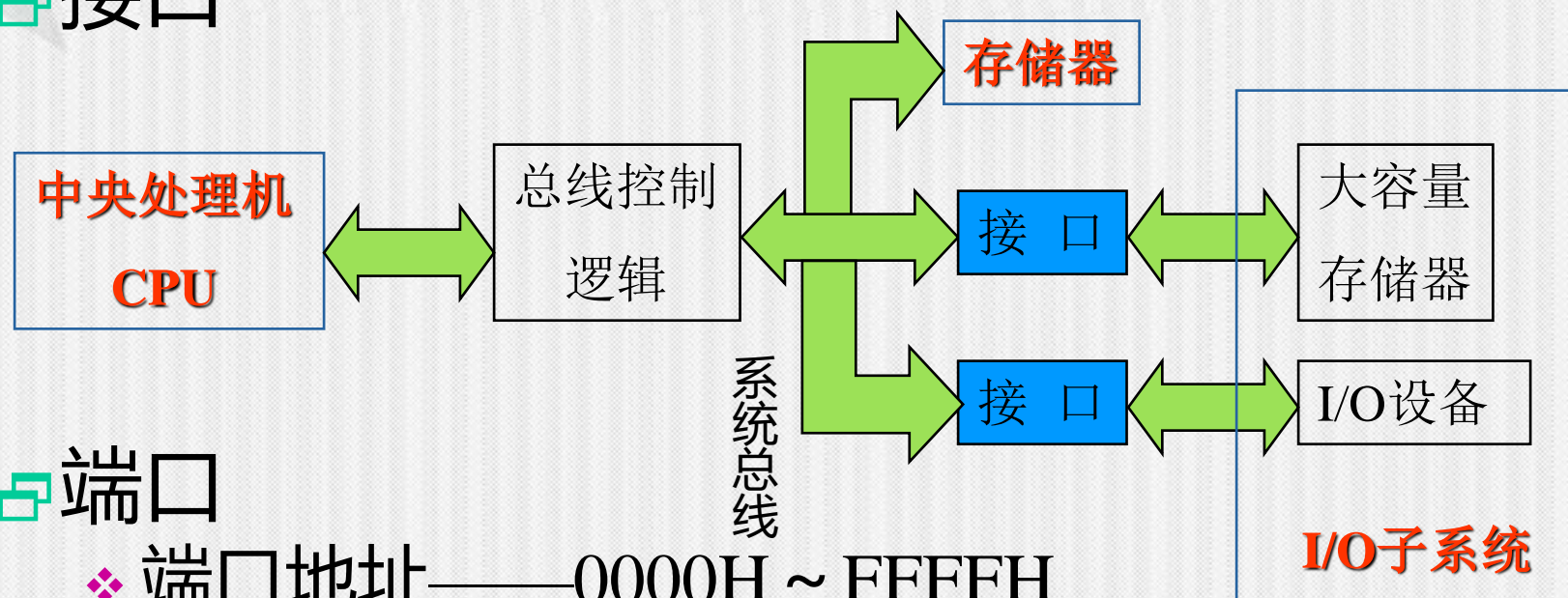
Assembly Language Programming

第五章

高级汇编语言程序设计

输入输出程序设计

接口



端口

- ❖ 端口地址——0000H ~ FFFFH
- ❖ 独立编址方式:与Memory地址独立
- ❖ 都是8位端口
- ❖ 端口分类: 数据端口, 状态端口, 控制端口

IN/OUT指令

分类:

- ❖ 长格式: 端口号00H~FFH可直接在指令中指定
- ❖ 短格式: 如果端口号 ≥ 256 , 端口号 \rightarrow DX

输入指令IN

格式: IN AL/AX, PORT/DX

举例

IN AL,25 ;

AL ← 25号端口的内容

IN AX,25 ;

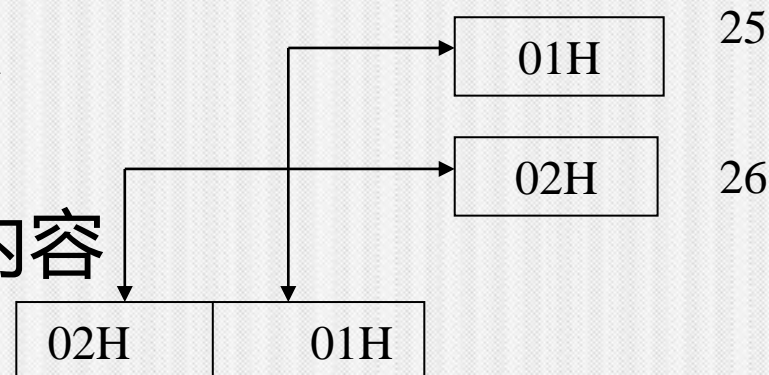
AX ← 25和26端口的内容

IN AL,DX ;

AL ← (DX)所指端口的内容

IN AX,DX ;

AX ← (DX)、(DX)+1所指端口的内容



输出指令

❏ 格式: `OUT PORT/DX, AL/AX`

❏ 举例:

`OUT 25,AL ;`

`(AL) → 25号端口`

`OUT 25,AX ;`

`(AX) → 25号端口和26号端口`

`OUT DX,AL;`

`AL → (DX)所指端口`

`OUT DX,AX;`

`(AX) → (DX)和(DX)+1二个端口`

举例：查询式输入输出

- ❏ 26H——input register
- ❏ 27H——output register
- ❏ 28H——status register(0-input ready, 1-output ready)
- ❏ 读入一个byte, 取反后再输出

ASM

Waitinput:

IN AL, 28h

TEST AL, 1

Jz waitinput

IN AL,26H

NOT AL

MOV BL,AL

Waitoutput:

IN AL,28H

TEST AL, 2

JZ Waitoutput

MOV AL,BL

OUT 27H,AL

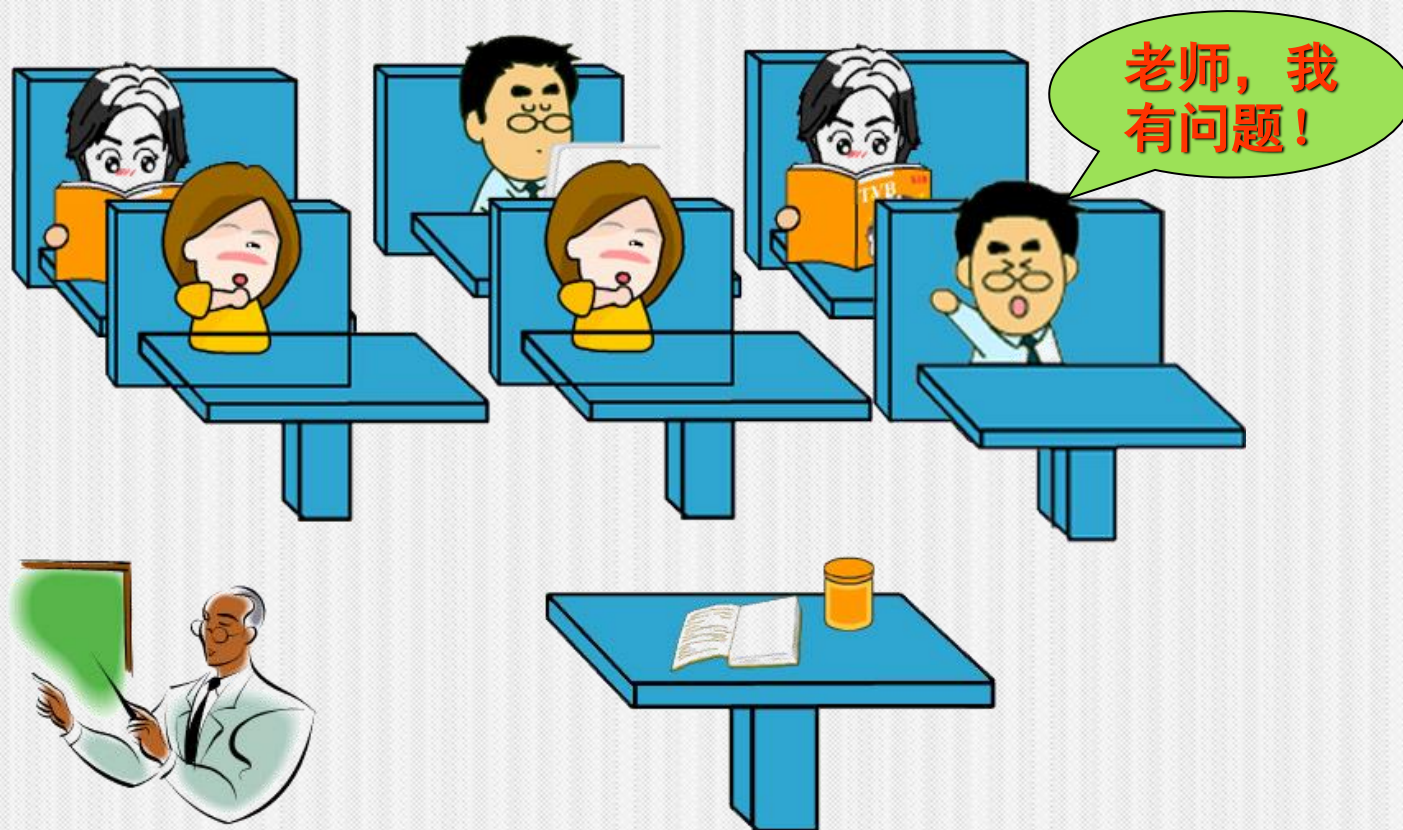
查询式I/O方式

ASM



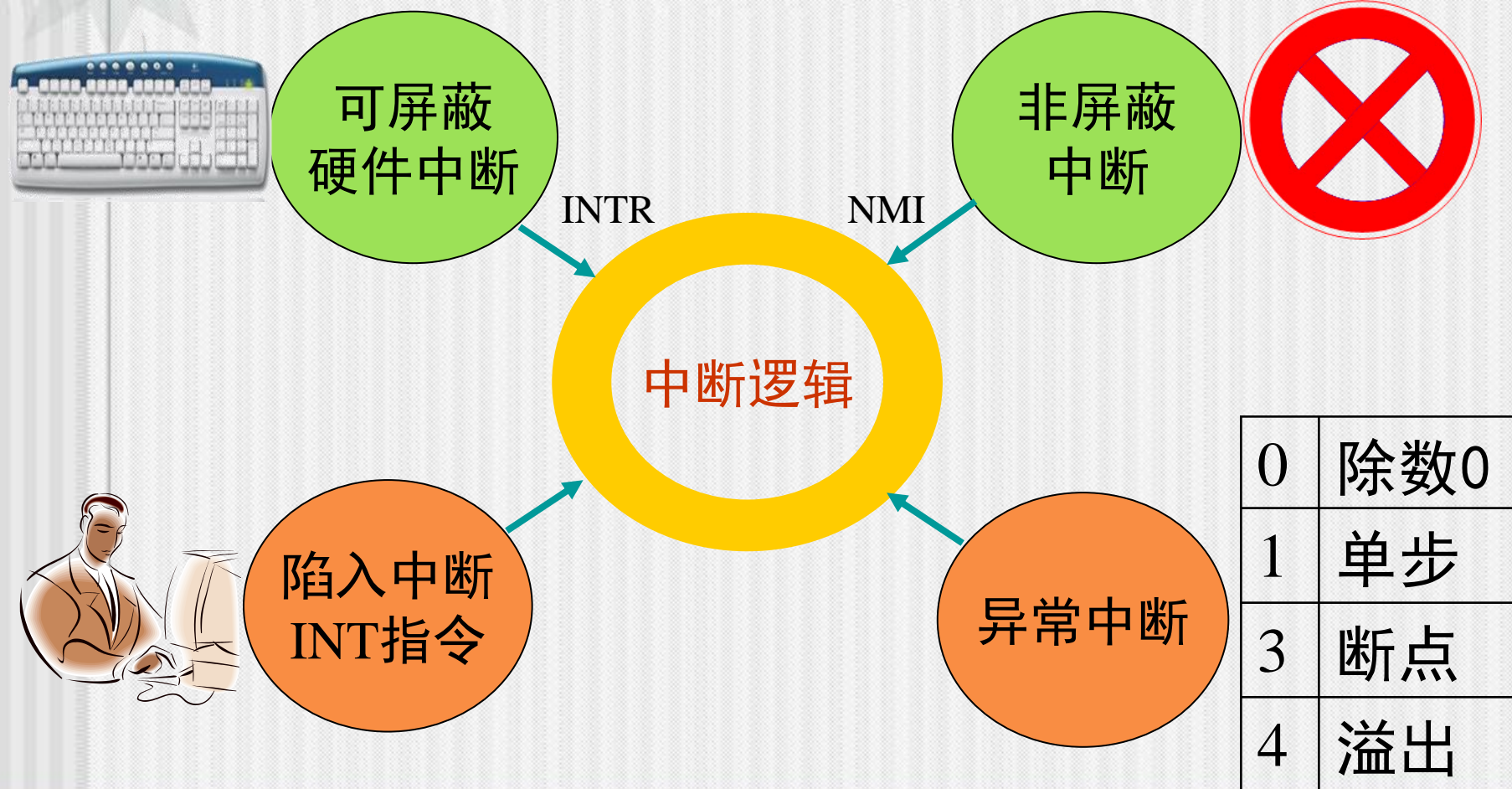
中断 (Interrupt)

ASM



中断源

中断源：引起中断的事件



中断屏蔽

IF



中断嵌套

老师，数据传输方式有哪些？

老师，中断是什么？

中断是一种数据传输方式.....

数据传输方式有.....四种！

.....中断方式的特点是效率高。

中断向量与中断类型码

❏ 中断服务子程序

- ❖ 每种中断都有与之对应的处理程序

❏ 中断向量

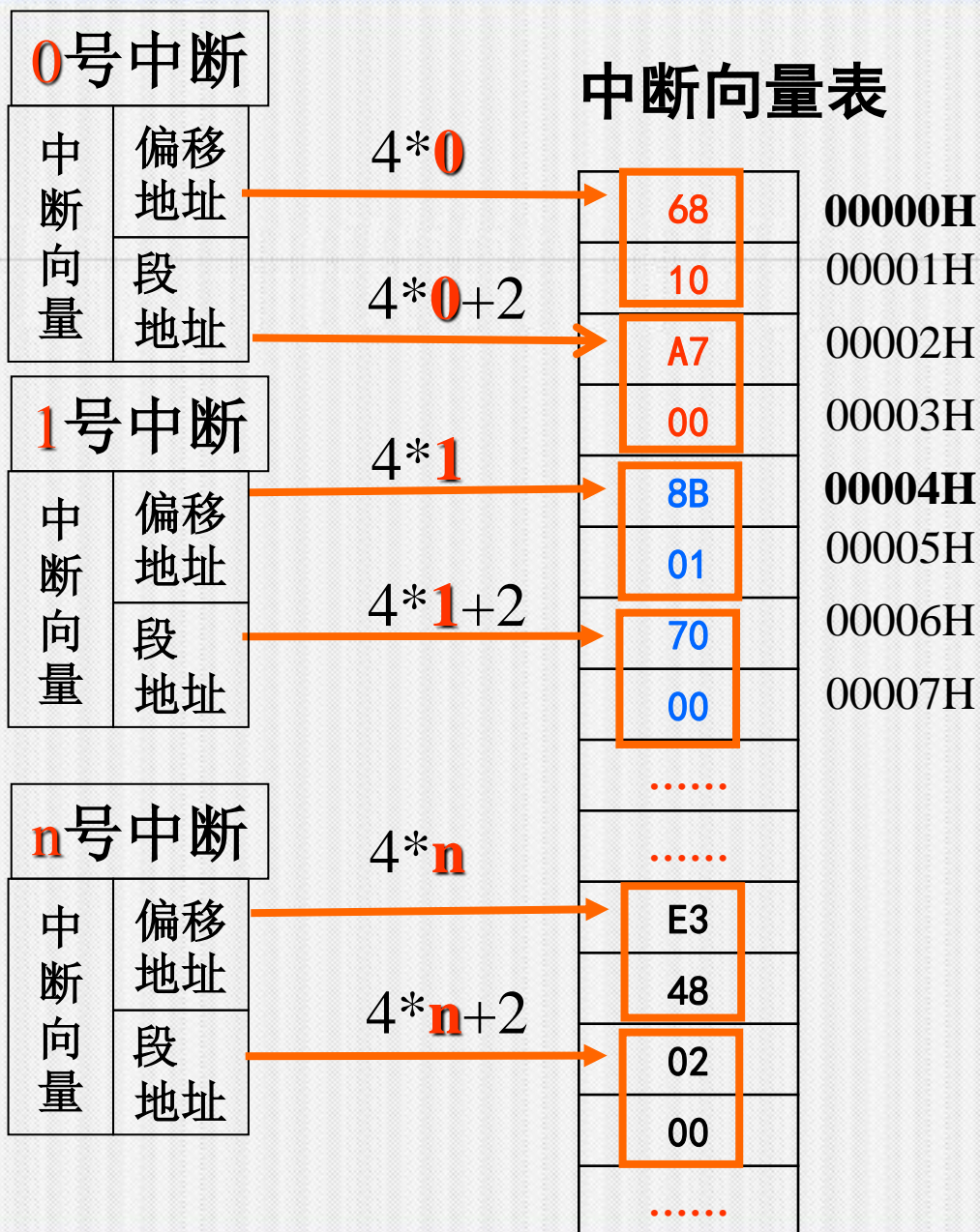
- ❖ 中断服务子程序的入口地址(16位偏移地址, 16位段地址)

❏ 中断向量表

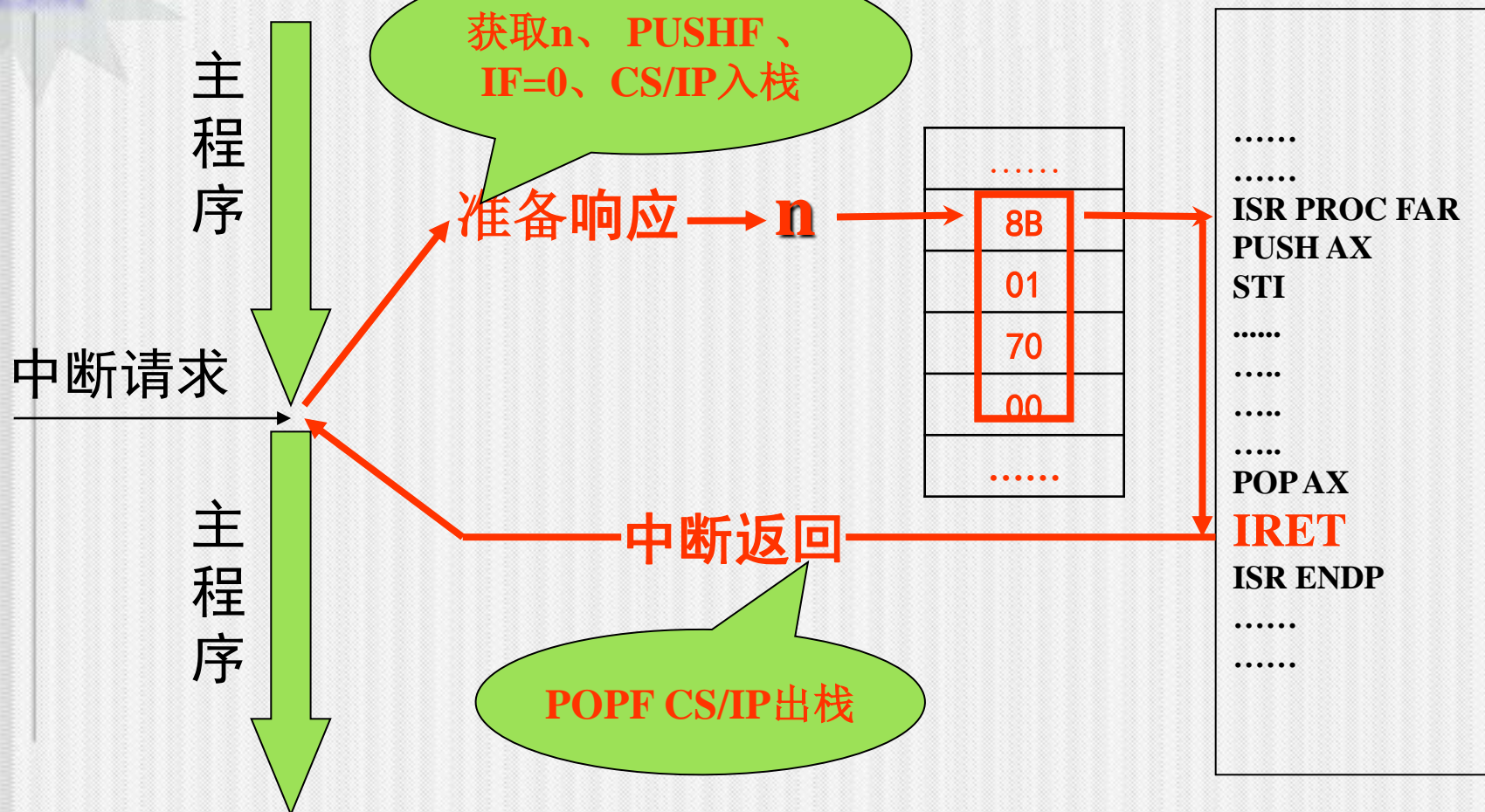
- ❖ 存放中断向量的表格。
- ❖ 256个, 00000H-003FFH, 1KB

❏ 中断类型码

- ❖ 表格的编号 n



中断过程



中断服务子程序

与一般子程序的差别:

- ❖ 中断服务子程序应为FAR
- ❖ 中断响应时 $IF=0$ ，子程序里一般应 $IF \leftarrow 1$
- ❖ 硬件中断处理程序，最后发中断结束EOI命令
- ❖ 返回为IRET而非RET
- ❖ 由系统进行调用

中断服务子程序的编写

ISR PROC **FAR**

PUSH AX

STI ;便于中断嵌套

.....

CLI

EOI (End Of Interrupt)

POP AX

IRET ;中断返回

ISR ENDP

保护现场

开中断

处理中断

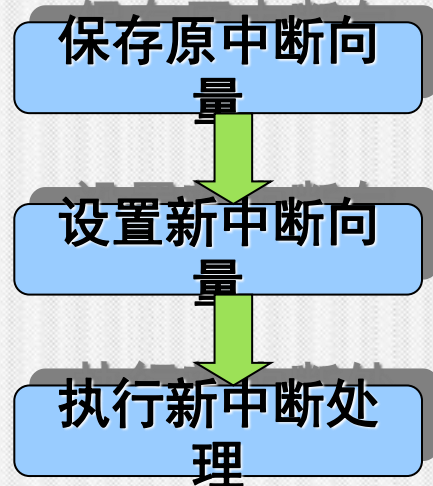
关中断

发中断结束命令

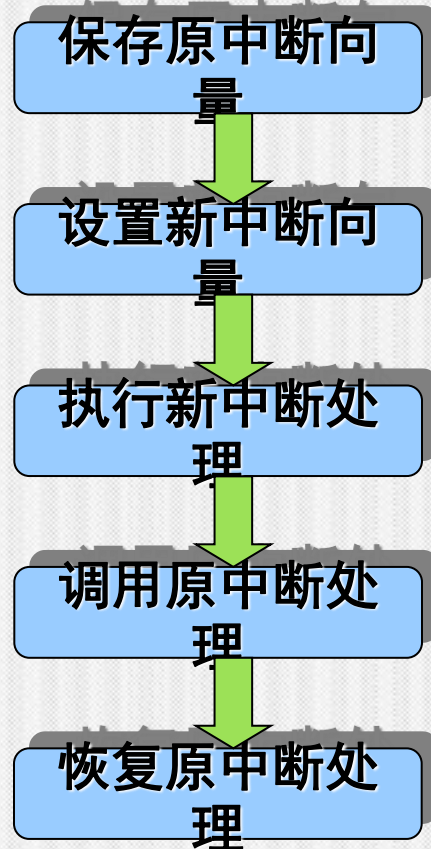
恢复现场

中断返回

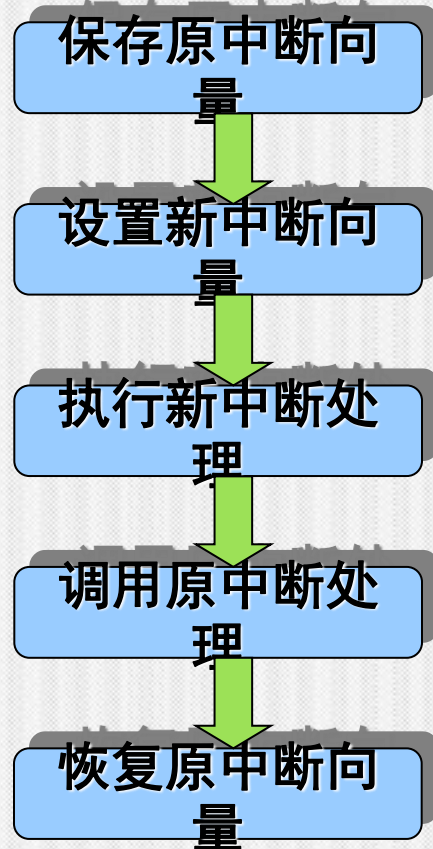
完整中断程序的编写



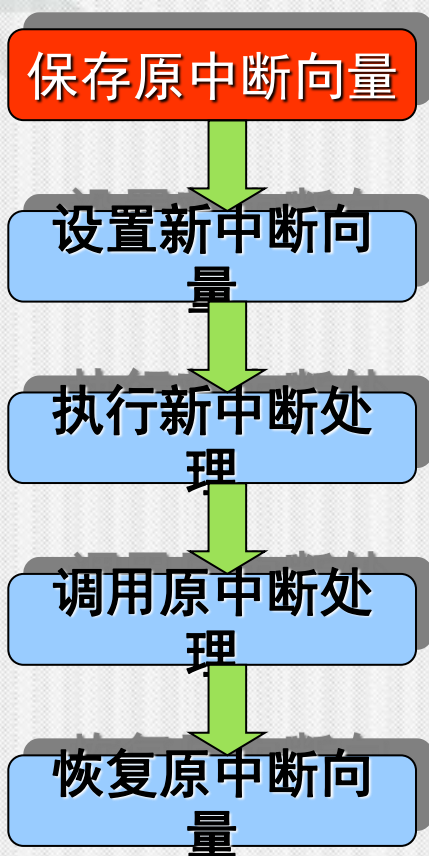
完整中断程序的编写



完整中断程序的编写



保存原中断向量



OLDISR DW ?,?

; ES = 0

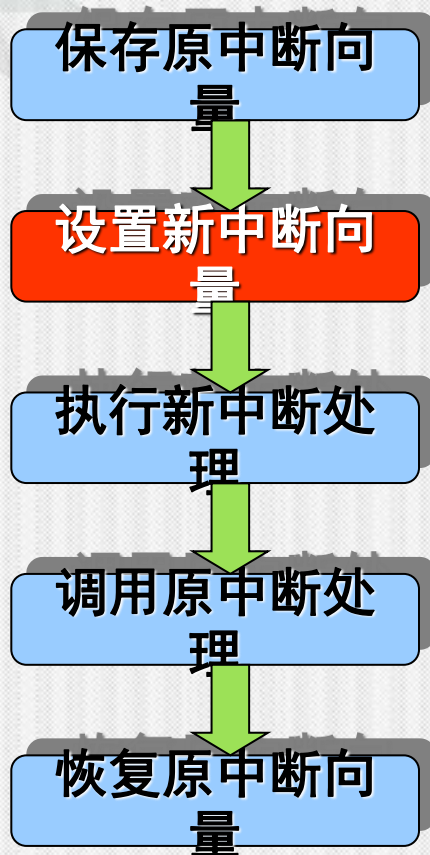
MOV AX, ES:[N*4]

MOV OLDISR[0], AX

MOV AX, ES:[N*4+2]

MOV OLDISR[2], AX

设置新中断向量

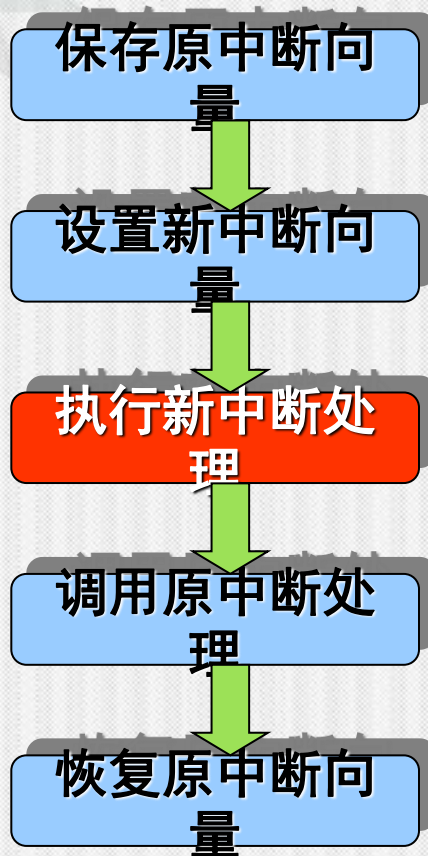


; ES = 0

MOV ES:[N*4], OFFSET ISR

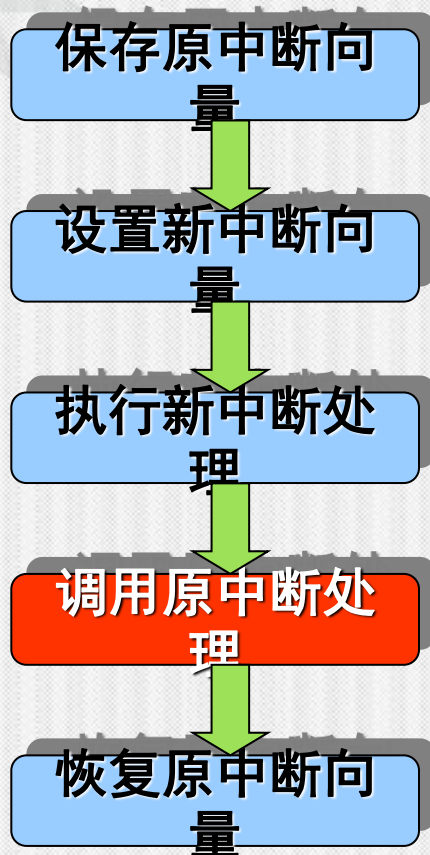
MOV ES:[N*4+2], SEG ISR

执行新中断处理



.....

调用原中断处理



中断过程:

.....

PUSHF

保护断点: PUSH CS; PUSH IP

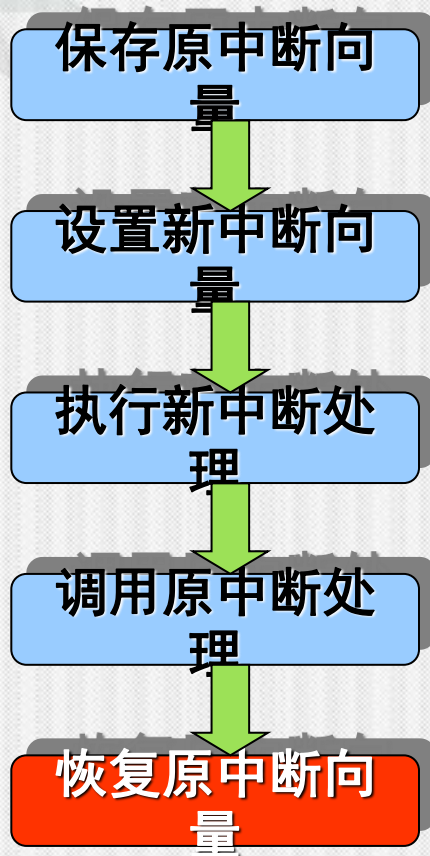
取中断向量, 并执行

OLDISR DW ?,?

PUSHF

CALL DWORD PTR OLDISR

恢复原中断处理



OLDISR DW ?,?

; ES = 0

MOV AX, OLDISR[0]

MOV ES:[N*4], AX

MOV AX, OLDISR[2]

MOV ES:[N*4+2], AX

举例：定时器实现

❏ 定时中断 — BIOS INT 08H

系统加电初始化后，定时器每隔约55毫秒发出一次中断请求。

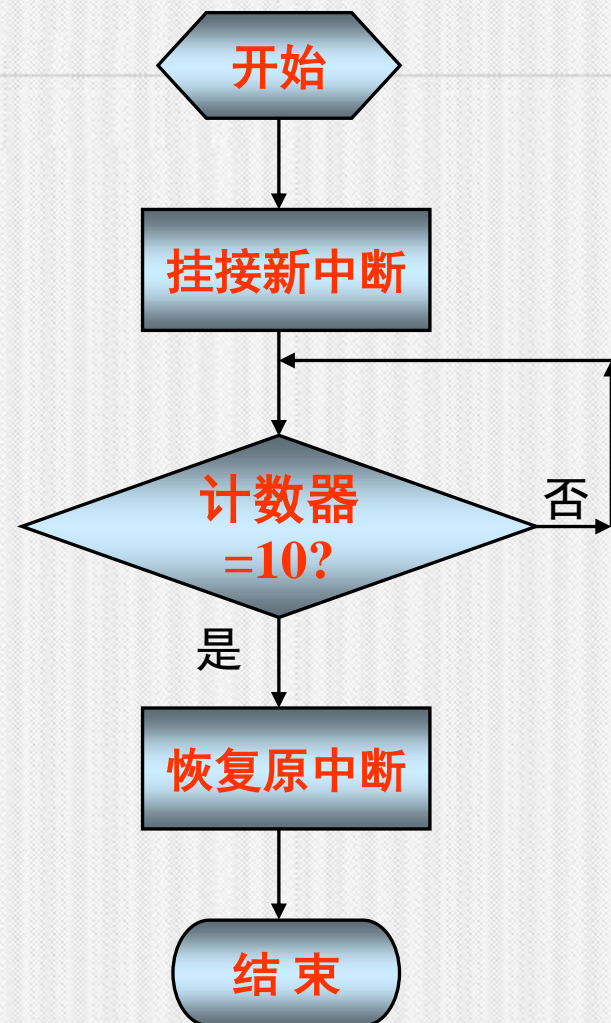
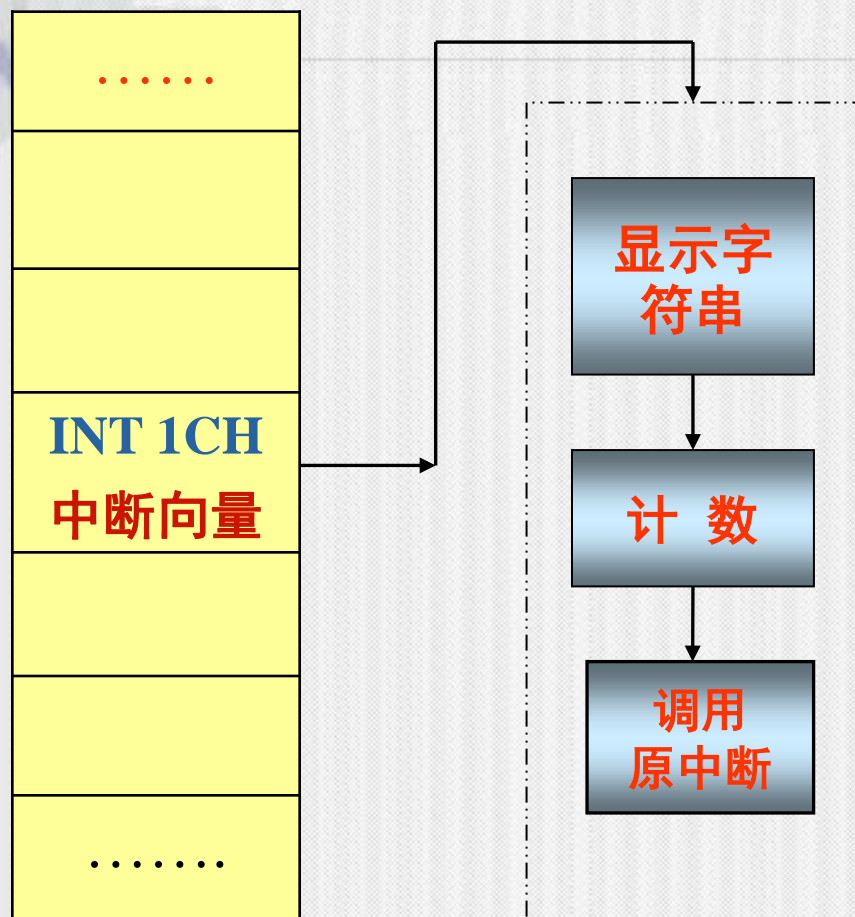
❏ INT 1CH：BIOS提供的8H号中断处理程序中有一条中断指令INT 1CH,所以每秒要调用到约18.2次1CH号中断处理程序。

❏ 例子

- ❖ 挂接INT 1CH，显示10次字符串
- ❖ 挂接INT 1CH，从30倒计时到0

中断服务子程序

主程序



DATA SEGMENT

STRING DB 'INT 1CH IS HOOKED! ',0DH,0AH,'\$'

OLDISR DW ?,?

TIMER DB 0

DATA ENDS

CODE SEGMENT

ASSUME CS:CODE,DS:DATA

START:MOV AX,DATA

MOV DS,AX

MOV AX,0

MOV ES,AX

MOV AX, ES:[1CH*4]

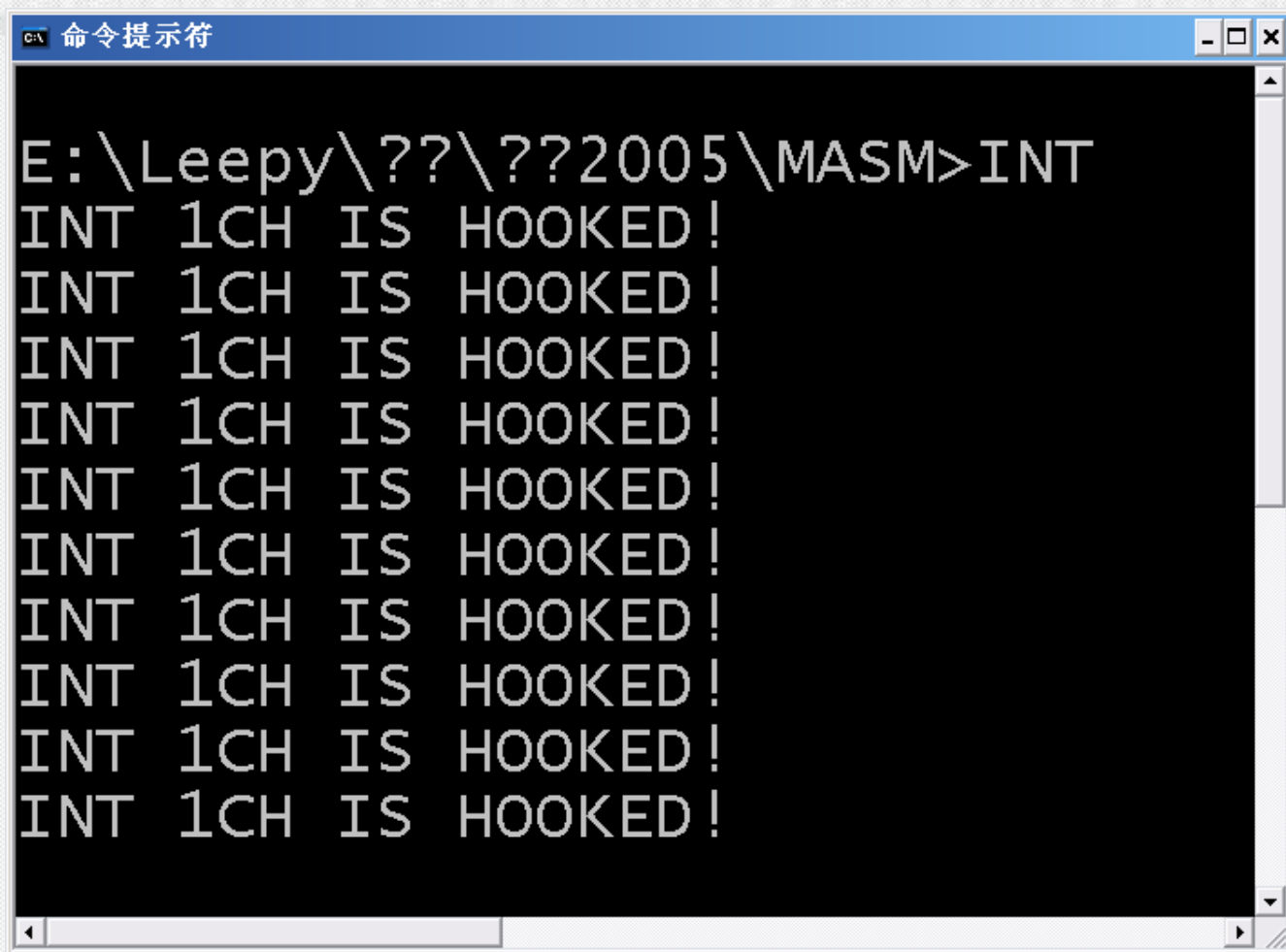
MOV OLDISR[0], AX

MOV AX,ES:[1CH*4+2]

MOV OLDISR[2], AX

保存原中断向
量

运行结果



```
命令提示符
E:\Leepy\??\??2005\MASM>INT
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
INT 1CH IS HOOKED!
```


倒计时程序

- ❏ 挂接INT 1CH, 从30倒计时到0
- ❏ Timer.asm

INTNO EQU 1CH

DATA SEGMENT

OLDISR DW ?,?

TIMER DB 100

COUNTER DW 30

ISDONE DB 0

DATA ENDS

CODE SEGMENT

ASSUME CS:CODE,DS:DATA

START:MOV AX,DATA

MOV DS,AX

MOV AX,0

MOV ES,AX

CLI; //关中断

MOV AX, ES:[INTNO*4]

MOV OLDISR[0], AX

保存原中断向量

MOV AX, ES:[INTNO*4+2]

MOV OLDISR[2], AX

设置新中断向量

MOV WORD PTR ES:[INTNO *4], OFFSET ISR

MOV WORD PTR ES:[INTNO *4+2], SEG ISR

STI

;WAIT HERE

WAITHERE:

等待执行新中断
处理

CMP ISDONE, 1

JNZ WAITHERE

EXIT:

;RESTORE

MOV AX,OLDISR[0]

MOV ES:[1CH*4],AX

MOV AX,OLDISR[2]

MOV ES:[1CH*4+2],AX

MOV AX,4C00H

INT 21H

;中断服务子程序

ISR PROC FAR

PUSH DX


PUSH AX

MOV AX,DATA

MOV DS,AX

STI

恢复原中断向量



```
;COUNT HERE
    INC TIMER
AGAIN:
    CMP TIMER, 1000/55 ;18
    JB DONE
    MOV TIMER,0
    ;打印
    MOV AH,2
    MOV DL,13
    INT 21H
```


;print time

MOV AX,COUNTER

MOV DL,10

DIV DL

MOV DH,AH

MOV DL,AL

MOV AH,2

ADD DL,30H

INT 21H

MOV DL,DH

ADD DL,30H

INT 21H

DEC COUNTER
JGE DONE
MOV ISDONE,1

DONE:

调用原中断处理

PUSHF
CALL DWORD PTR OLDISR
CLI
POP AX
POP DX
IRET ;中断返回

ISR ENDP
CODE ENDS
END START

汇编语言程序设计

Assembly Language Programming

第六章

32位指令与混编

6.1 32位指令

16位指令系统从两个方面向32位扩展

- (1) 支持32位操作数
- (2) 支持32位寻址方式

有些指令扩大了工作范围，或指令功能实现了向32位的自然增强

寄存器组

8个32位通用寄存器：

EAX EBX ECX EDX

ESI EDI EBP ESP

6个16位段寄存器：

CS SS DS ES FS GS

32位指令指针寄存器：EIP

32位标志寄存器：EFLAGS

其他的32位系统用寄存器

在原有16位寄存器
基础上扩展成为32位

寻址方式

32位有效地址 =

基址寄存器 + (变址寄存器 × 比例) + 位移量



- 基址寄存器——任何8个32位通用寄存器之一
- 变址寄存器——除ESP之外的任何32位通用寄存器之一
- 比例——可以是1 / 2 / 4 / 8
- 位移量——可以是8 / 32位值

数据传送

ASM

`mov eax,ebx`

;32位操作数

`mov ax,[ebx]`

;16位操作数, 32位寻址方式

`mov eax,[ebx]`

;32位操作数, 32位寻址方式

将立即数压入堆栈

PUSH i8/i16/i32

;把16位或32位立即数i16/i32压入堆栈。若是8位立即数i8, 经符号扩展成16位后再压入堆栈

push word ptr 1234h ;压入16位立即数

push dword ptr 87654321h ;压入32位立即数

call helloabc

add esp, 6 ;平衡堆栈

通用寄存器全部进出栈

PUSHA

;顺序将AX/CX/DX/BX/SP/BP/SI/DI压入堆栈

POPA

;顺序从堆栈弹出DI/SI/BP/SP/BX/DX/CX/AX
(与PUSHA相反)

;其中应进入SP的值被舍弃，并不进入SP，SP
通过增加16来恢复

符号扩展和零位扩展

MOVSX r16,r8/m8

;把r8/m8符号扩展并传送至r16

MOVSX r32,r8/m8/r16/m16

;把r8/m8/r16/m16符号扩展并传送至r32

MOVZX r16,r8/m8

;把r8/m8零位扩展并传送至r16

MOVZX r32,r8/m8/r16/m16

;把r8/m8/r16/m16零位扩展并传送至r32

```
mov bl,92h
```

```
movsx ax,bl           ;ax=ff92h
```

```
movsx esi,bl          ;esi=ffffff92h
```

```
movzx edi,ax          ;edi=0000ff92h
```


6.2 混合编程

- ❏ 多种程序设计语言间，通过相互调用、参数传递、共享数据结构和数据信息而形成程序的过程就是混合编程
- ❏ 程序的大部分采用高级语言编写，以提高程序的开发效率；在某些部分，利用汇编语言编写，以提高程序的运行效率

混合编程方法

❏ 嵌入式汇编——

- ❖ 在C/C++语言中直接使用汇编语言语句，
- ❖ 简洁直观、功能较弱

❏ 模块连接——

- ❖ 两种语言分别编写独立的程序模块，分别产生目标代码OBJ文件，然后进行连接，形成一个完整的程序
- ❖ 使用灵活、功能强，要解决参数传递问题

嵌入汇编语言

格式

`__asm { 指令 }`

举例

```
int power2(int num,int power)
```

```
{ __asm  
{
```

```
    mov eax,num
```

```
    mov ecx,power
```

```
    shl eax,cl
```

```
} // 返回 EAX=EAX×(2^CL)
```


6.3 查看C++对应汇编代码

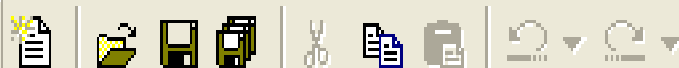
```
#include "stdafx.h"
```

```
int fun(int x,int y)
{
    int z;
    z=x+y;
    return (z); /*返回运算结果*/
}
```

```
int main(int argc, char* argv[])
{
    int c;
    c=fun(3,100);
    printf("%d",c);
    return 0;
}
```


aa - Microsoft Visual C++

文件(F) 编辑(E) 查看(V) 插入(I) 工程(P) 组建(B) 工具(T) 窗口(W) 帮助(H)



[Globals] (All global mem

编译 [aa.cpp] Ctrl+F7
组建 [aa.exe] F7
全部重建
批组建...
清除

开始调试(D)

远程连接调试程序...

! 执行 [aa.exe] Ctrl+F5

移除工程配置(R)...

配置...

配置文件...

point for the console applicat

GO F5

{ } Step Into F11

*{ } Run to Cursor Ctrl+F10

附加到当前进程(A)...

// aa
//

#incl

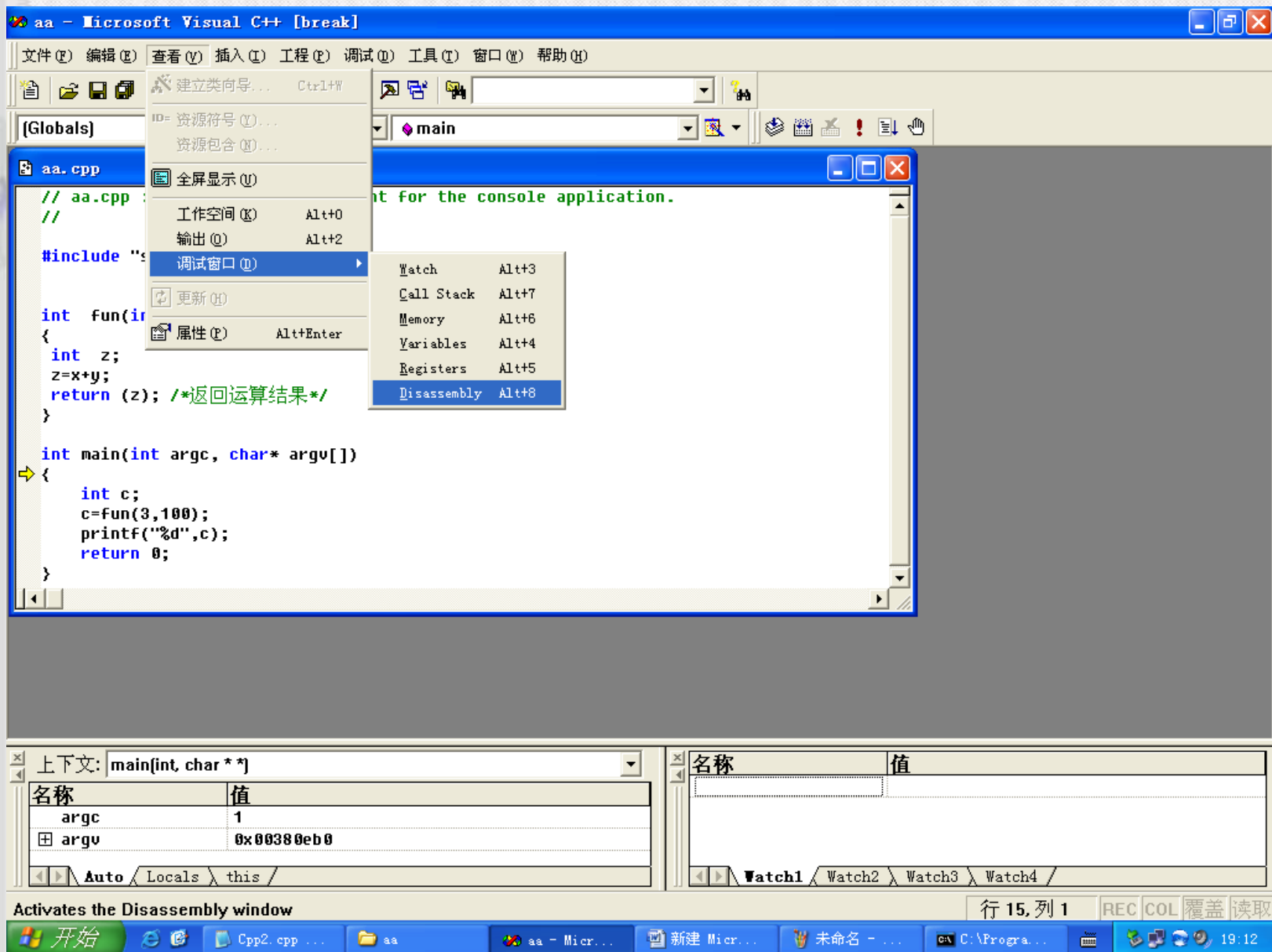
int

{

int

Z=X+Y;

return (z); /*返回运算结果*/



aa - Microsoft Visual C++ [break]

文件(F) 编辑(E) 查看(V) 插入(I) 工程(O) 调试(D) 工具(T) 窗口(W) 帮助(H)

[Globals] [All global members] main

aa.cpp

// aa.cpp : Defines the entr
//

#include "stdafx.h"

int fun(int x,int y)

{
int z;
z=x+y;
return (z); /*返回运算结果*
}

int main(int argc, char* arg

{
int c;
c=fun(3,100);
printf("%d",c);
return 0;
}

Disassembly

```

15:  {
    0040D6F0  push     ebp
    0040D6F1  mov      ebp,esp
    0040D6F3  sub      esp,44h
    0040D6F6  push     ebx
    0040D6F7  push     esi
    0040D6F8  push     edi
    0040D6F9  lea      edi,[ebp-44h]
    0040D6FC  mov      ecx,11h
    0040D701  mov      eax,0CCCCCCCch
    0040D706  rep stos dword ptr [edi]
16:  int c;
17:  c=fun(3,100);
    0040D708  push     64h
    0040D70A  push     3
    0040D70C  call     @ILT+5(fun) (0040100a)
    0040D711  add      esp,8
    0040D714  mov      dword ptr [ebp-4],eax
18:  printf("%d",c);
    0040D717  mov      eax,dword ptr [ebp-4]
    0040D71A  push     eax
    0040D71B  push     offset string "%d" (0040
    0040D720  call     printf (00401060)
    0040D725  add      esp,8
19:  return 0;

```

上下文: main(int, char **)

名称	值
argc	1
argv	0x00380eb0

Auto Locals this

名称 值

Watch1 Watch2 Watch3 Watch4

就绪

开始 Cpp2.cpp ... aa aa - Micr... 新建 Micr... 未命名 - ... C:\Progra... 19:13